Introspective Service Interface Synthesis in Business Networks

Fuguo Wei, Alistair Barros, and Chun Ouyang

Queensland University of Technology, Brisbane, Australia
{f.wei,alistair.barros,c.ouyang}@qut.edu.au

Abstract. Service mismatches involve the adaptation of structural and behavioural interfaces of services, which in practice incurs long lead times through manual, coding effort. We propose a framework, complementary to conventional service adaptation, to extract comprehensive and semantically normalised service interfaces, useful for interoperability in large business networks and the Internet of Services. The framework supports introspection and analysis of large and overloaded operational signatures to derive focal artefacts, namely the underlying business objects of services. A more simplified and comprehensive service interface layer is created based on these, and rendered into semantically normalised interfaces, given an ontology accrued through the framework from service analysis history. This opens up the prospect of supporting capability comparisons across services, and run-time request backtracking and adjustment, as consumers discover new features of a service's operations through corresponding features of similar services. This paper provides a first exposition of the service interface synthesis framework, describing patterns having novel requirements for unilateral service adaptation, and algorithms for interface introspection and business object alignment. A prototype implementation and analysis of web services drawn from commercial logistic systems are used to validate the algorithms and identify open challenges and future research directions.

Keywords: service, service interface synthesis, service adaptation, business networks

1 Introduction

Services have proliferated over recent years through the transformation of businesses into global networks, and the surge of consumer-based, on-demand "apps", driving a new wave of enterprise services. As a result, services are becoming the established means of ensuring that companies lower the total cost of ownership of their systems, focusing on core competencies, and leveraging capabilities through loosely coupled collaborations with partners [1]. However, the rapid growth of services becoming available also poses challenges for companies aiming to capitalise on these and integrate them into business processes. The degree of data heterogeneity and the rate of evolution of functional capabilities of services are

outpacing the conventional means to adapt and interoperate services in diffuse network settings, scaled out to the Internet.

Research into service adaptation has been ongoing, addressing the problems of reconciling mismatches of service interfaces, encountered in distributed, heterogeneous settings. Structural mismatches refer to the incompatibilities in operational signatures (data parameters and types) occurring on a syntactic level, i.e. type-compatibility of data parameters, or a semantic level, i.e. the meaning of parameters. Behavioural interface mismatches relate to incompatibilities of interaction sequences on services, i.e. message exchange sequences, or protocols, between services. Managing structural and behavioural mismatches requires costly adaptation of interfaces at design-time so that services can be integrated. To date, many techniques have been proposed for supporting semi- or fully-automated derivation of adapters which enable interactions over structural and behavioural mismatches, including the use of semantic annotation of interfaces based on ontologies [2, 3]. However, these techniques often result in too much reliance on service providers to gain an understanding of the intricate details of service interfaces so that service consumers or third-parties can feasibly build or derive the necessary service adapters. Thus, they incur significant lead times and costly maintenance to yield service adapters, and their productivity in the context of dynamic service growth on the scale of the Internet remains uncertain.

This paper proposes a new and complementary strategy to conventional service adaptation, whereby sufficient knowledge of service interfaces can be unilaterally analysed by service consumers. Specifically, the paper proposes a service interface synthesis framework, where service interfaces are *introspected*, for both *structural* and *behavioural* aspects. Services, especially of commercial, business applications, typically have large and complex operations, seen through a large number of parameters, requiring "in-house" knowledge about the allowable invocations (subsets of parameters). In essence, the introspection approach we develop strives to extract the core artifacts of operations, which for business applications corresponds to business objects (e.g. purchase order, customer, mortgage, insurance claim). We have shown in [4] that operational overloading arises, in large part, because of multiple business objects or business object specializations or variants present in the same operations. Typical examples like the type of goods, the sources of approval, prior or ad-hoc contractual arrangements, payment agreements, and special delivery provisions (such as insurance or third-party transportation), lead to many parameters, each part of different business objects and specializations, present in the same operations. This makes it difficult to determine which subsets of parameters to include an operation invocation. The introspection technique of the framework essentially elicits knowledge of the business objects, their attributes, associations and specialisations. This can lead to the refactoring of a new interfaces, aligned to business object access operations (queries and updates), allowing a simplified and comprehensive understanding of service interface operations. Such a layer makes it possible to semantically align different services, based on their elicited artefacts, using a common ontology. It also supports enriched interactions such as evolving requests based on insights

of the capabilities of similar services, for example a shipping request may be evolved to include insurance coverage if the consumer queries another, similar service and discovers that such a feature is offered by it as part of its shipping capabilities. Such cross-checks could be applied at run-time by consumers, i.e. "shopping" for service features.

The paper firstly provides novel insights and patterns motivating the need for unilateral service interface synthesis (in Sect. 2). We next elaborate on the key steps of the synthesis and develop detailed insights into its most novel feature in service interface synthesis (in Sect. 3). The paper focusses on the introspection algorithms for structural and behavioral aspects only, bringing into view the basis for addressing backtracking and normalisation requirements for future work. Sect. 4 shows the implementation of the framework using FedEx open shipping services and reveals some open issues. Finally, after a review of related research efforts (in Sect. 5), Sect. 6 concludes the paper and outlines the future work.

2 Requirements Analysis

This section presents four patterns capturing service interoperability problems in the context of service interactions in global business networks. Unlike the previous contributions of patterns [5], these patterns do not assume detailed knowledge of service interfaces by service consumers.

Pattern 1 (Interaction without full structural interface knowledge)
Description Following the discovery of a service and its structural interface (e.g. a WSDL interface), a service consumer starts to interact with this service (provider). The interaction involves invoking any operation on the service, passing the required input (e.g. message documents) and receiving the output as a result of the invocation. The cognition of the interaction is on invoking the service, and therefore on a detailed understanding of the called service's structural interface, as opposed to a receiving invocation (e.g. as in a call-back invocation). Receiving invocations are simply a reverse direction of this pattern's focus and are therefore conceptually covered. The scope of the interaction is on single invocation of an operation (invocation of multiple operations in a single interaction context can be achieved through the design of a single operation which controls invocation of other the operations). Reciprocal invocations across the service consumer and provider require further knowledge of the behavioural interface of the service, and are considered in Pattern 2.

Examples In a purchase order process, to send a purchase order request, the SAP purchasing service has around 81 parameters and most of them have a complex data type. These parameters can be combined in a different way for different invocations of the same operations on the service. For example, the material master type (goods ordered) leads to the applicability of different pa-

rameters such as delivery and storage plant. Similarly, the Oracle PeopleSoft purchasing service has 38 complex parameters.

Issues/design choices Although structural interface knowledge can be obtained through service discovery mechanisms (e.g. through service repositories implementing service description languages [6], operations can be complex and overloaded, leading to ambiguities as to what the valid invocations are. This overloading arises because a service has multiple variants, as in a purchase ordering service procuring small/high charge, biodegradable, flammable etc. goods, leading to widely varying parameters sets on innovations of the same service operation. Therefore, a service consumer needs refined insights as to what valid combinations of parameters are required for all possible valid invocations of the same operation, for each operation of the service. This issue is orthogonal to a semantic understanding of the parameters and guidance mechanisms for service interactions based on semantic assumptions [7].

Solution A proposed strategy is ad-hoc discovery of service operational knowledge based on introspection. Since the only knowledge of structural interfaces contains ambiguities due to operation overloading, a trial/error introspection can be adopted. Given the input and output parameters of an operation on a service, unique combinations of input and output parameter sets could be derived, and each can be used to invoke the service using sample data values. Accordingly, the set of valid operation invocations can be determined. A particular issue is that valid combinations could be subsets of the core set of operation invocations, i.e. they could be combined in core operations for comprehensive output from the service. Therefore, a second pass of the valid invocations needs to be applied to determine which ones are covered by "maximal" invocation sets. The final list of recommended invocations requires designer confirmation due to semantic interpretations which cannot be derived automatically.

Pattern 2 (Interaction without full behavioural interface knowledge)
Description Following the discovery of a service and its behavioural interface (e.g. a WS-BPEL abstract process of a service), a service consumer needs to interact with it through several interactions in the sequential order required by the interface. Each interaction involves valid invocations of service operations which are resolved through solutions addressed in pattern 1. The cognition of the interaction is on invocations at the provider side and the valid sequences, or protocols, of interactions, i.e. sending messages to the service, receiving message from the service. In other words, the cognition is on the provider side protocol. Obviously, both consumer and provider protocols need to be integrated in order for reciprocal message exchanges to take place, however the consumer side is the reverse direction of this pattern's focus and is conceptually covered.

Examples A supplier service wants to call "AskforDelivery" to a carrier service. However, it does not know what the steps are to ask for delivery. For example,

the carrier service may expect the purchase order details and letter of credit to be received before "AskforDelivery" is invoked.

Issues/design choices The availability of behavioural interfaces are not guaranteed in practice [8]. Even if they are available, behavioural interfaces, as structural interfaces, present ambiguities because of the presence of service variants (as discussed in pattern 1). Different variants may lead to differences in service interactions, and all of which are optional and determined through run-time as to which choice of interactions is required. Therefore, the service consumer cannot easily determine which particular part of the behavioural protocol applies for interacting with a service.

Solution A protocol discovery process is needed. It is necessary to have a mechanism to guide services to send messages in the right sequence. The mechanism can make "dryrun" calls to test the protocols of service interactions at design time. Once the protocols are identified, the mechanism can guide the services to interact at run-time.

Pattern 3
(Interaction backtracking based on structural interface learning)

Description A service consumer learns new structural interface knowledge about a provider's service following an interaction with a similar service from another provider. Specifically, it makes a request to the first provider's service (e.g., a shipping provider such as FedEx), and then makes a similar request to the second provider's (e.g., UPS) service. Based on the interaction with the second provider, it discovers a new feature of the request, and accordingly it updates the request with the first provider. In turn, it may learn further knowledge about the service request through the update with the first provider, and it may update the request with the second provider. In other words, it progressively learns about new features from similar service interactions with different providers (much like shopping for products where exposure to different products reveals a new understanding of the ideal feature set). This backtracking can be generalised to n providers, not just two.

Examples A purchasing service asks for quotation from both supplier S_1 and supplier S_2. It sends two invocations to the two suppliers with messages such as "itemName, modelName, and supplierName" via a standard call "RequestQuotation" (i.e. a built-in interface of the purchasing service). The invocation is accepted by S_1, but it is rejected by S_2 due to interface mismatches. The mismatches can be, for example, S_2 asks for an additional parameter "Quantity". In this instance, a central mechanism can be used to accumulate the service consumer's knowledge of service interactions and make possible updates to previous interactions. The mechanism may propose the new parameter "Quantity" to S_1, which could be an optional parameter in S_1's interface set. With "Quantity", S_1 may give a better quotation (e.g., discounted price).

Issues/design choices There are three issues with dealing with learning and backtracking structural interface knowledge. The first is that different parame-

ters can be the same of semantic type. For example, a parameter combination (a, b, c, f) may be semantically equivalent to the combination (x, y, z, ω). The challenge to be addressed is to match the semantic equivalence. The second issue is to handle a request to a service without committing it. New structural interface knowledge learnt is tried on a service and the response from the service is then analysed. Depending on the response, the process may go back and forth several times. The request should not be committed until the new knowledge is ultimately accepted or rejected. The last issue is to avoid livelock. In other words, the "back and forth" process should not be endless and a decision point where the backtracking process can stop should be made.

Solution A proposed strategy is a backtracking mechanism. It incrementally learns structural knowledge from one service and then goes back and forth to apply the knowledge to the similar requests to other services. The mechanism needs to address the aforementioned issues in the "Issues/design choices" section. In particular, a semantic type matching mechanism is needed to map equivalent parameter combinations. A N-phase commit strategy is to be proposed to dryrun and test the new structural interface knowledge and to commit the real invocation with transactional effects in the end. There is also a need to have livelock detection mechanism to prevent an endless "back and forth" process.

Pattern 4
(Interaction backtracking based on behavioural interface learning)
Description Similarly as structural interface backtracking, a service consumer may learn new behavioural interface (i.e service interaction protocols) knowledge from the second service provider and then update the request with the first provider and so forth. Similarly, this "back and forth" mechanism can be generalised to n providers, not just two.

Examples A client interacts with two air ticket booking services. The client may learn that, before making a booking, the second service asks to enter frequent flyer details after the client is authenticated. However, the details were not asked in the first service as the step is considered optional. It is possible to backtrack and update the first service request, adding the frequent flyer information.

Issues/design choices There are similar issues as described in the section "Issues/design choices" of Pattern 3, in the context of behavioural interface, i.e. semantic matching of operations and sequences, request without commitment to support backtracking of requests and corresponding action sequences, and livelock detection.

Solution The key issue that arises from behavioural interfaces is the choices of action sequences corresponding to different interaction sequences required for different services. The moments of choice need to be carefully demarcated so that they can be used to support reasoning related to semantic matching of operations and sequences, request without commitment to support backtracking of requests and corresponding action sequences, and livelock detection. The behaviour of an interface can be specified in the form of a lifecycle of service operations using, for example, state transition models.

3 Service Interface Synthesis

3.1 Overview

To address the service adaptation challenges in business networks presented in the previous section, we propose a service interface synthesis framework. Essentially, the framework is comprised of two modules (as shown in Fig. 1). The first is the service interface analysis, which analyses service structural interfaces and discovers the order to invoke operations of a service. The second module is the service interface normalisation and it normalises interfaces revealed between similar services.

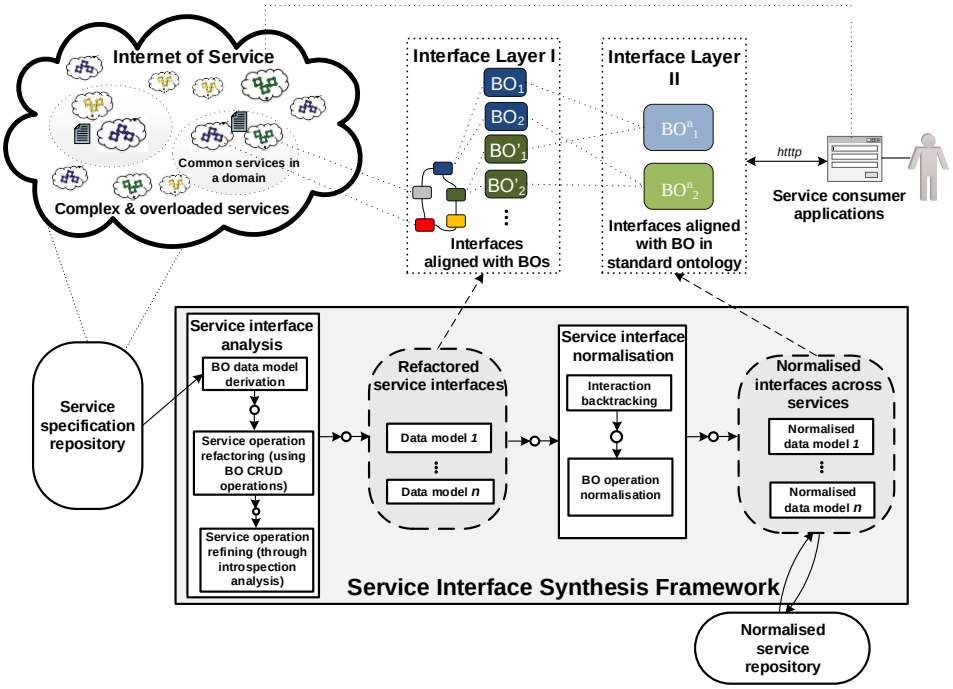

Fig. 1. An overview of the service interface synthesis framework.

The service interface analysis module has three components. The *BO data model derivation* component analyses the input and output parameters of operations on a service and map them to a business object (BO) data model. Services, in essence, focus on how resource states are addressed and transferred. This research argues that the business object is the primary resource manipulated by services in the context of global business networks. Therefore, the analysis is carried out based on the notion of business object. As a result of this component, complex service interfaces are refactored as the BO data model, which presents all business objects (and the relations among them) implied in a service. The *Service operation refactoring* component takes a BO data model as the input,

and then maps operations provided by a service to four generic operations of the business objects in the BO data model. In addition, we also generate the protocols for each CRUD operation. In this research, we propose an abstract business object with four generic operations and they are CREATE, READ, UPDATE, and DELETE (CRUD). The abstract business object is the parent of all business objects. As an output, these service specific BO CRUD operations form the interface layer 1. Complex and overloaded interfaces of a service are encapsulated, simplified. Having these structured interfaces, unique combinations of input parameter sets could be easily derived. The *Service operation refining* component then invokes and introspects the service using sample data values in order to determine the set of valid invocations.

Similarly, the normalisation module also consists of two components. The *Interaction backtracking* is proposed to apply knowledge learnt from other service providers and refine the requests from service consumers. The *BO operation normalisation* component normalises the BO data models and CRUD protocols of business objects implied in services that offer a similar capability. As an output, it produces the second interface layer, where structural interface and CRUD protocols of business objects from heterogeneous services are normalised. The normalised interfaces are then stored as references in the *Normalised service repository* for service adaptation.

As the first step of this study, this paper focuses on the first two components of the service interface analysis module.

3.2 Data Model

Definition 1 (Service data model). A service is a tuple (OP, BO, ξ). OP consists of a number of operations provided by s. BO is the set of business objects implied in s. $\xi \subseteq BO \times BO$ captures the dependency relations between objects in BO, i.e. for any two objects $(bo, bo') \in \xi$, bo depends on bo'. ξ is a transitive relation, and (BO, ξ) forms a directed graph. □

Definition 2 (Operation and parameter). Let OP be a set of operations and op any operation in OP. $\mathcal{N}(op)$ specifies the name of op, $\mathcal{I}(op)$ the set of input parameters, and $\mathcal{O}(op)$ the set of output parameters used by op.

Let P be a set of parameters, p any parameter in P, and $P' = P \backslash \{p\}$. $\mathcal{N}(p)$ specifies the name of p, $\gamma(p) \in \{primitive, complex\}$ whether p is of a primitive or a complex type, $type(p)$ the type of data (e.g. string, LineItem) carried by p, and $nest(p) \in 2^{P'}$ the set of parameters nested in p where $nest(p) = \varnothing$ if and only if $\gamma(p) = primitive$. □

Definition 3 (Business object). Let BO be a set of business objects, bo any object in BO, and OP a set of operations. $\mathcal{N}(bo)$ specifies the name of bo, $key(bo)$ the unique identifier of bo, $oprt(bo) \in 2^{OP}$ the set of operations applied to bo, and $\mathcal{A}(bo)$ the set of attributes associated with bo. For each attribute $a \in \mathcal{A}(bo)$, $\mathcal{N}(a)$ is the name of a and $type(a)$ is the type of data carried by a. □

Structural input and output interfaces of operations on a service are mapped to a business object based service data model (BO data model). Fig. 2 presents a generic BO data model. In this model, there are four concrete business objects (there could be more in a real example) and they are mapped from the $\mathcal{I}(op_1)$ and $\mathcal{O}(op_1)$. For a parameter p, if $\gamma(p)$ is *complex* (i.e., user defined), it is possibly mapped to a business object. For example, p_1 is mapped to BusinessObjectA. Each p in $nest(p)$ is then mapped to an attribute of the business object. For instance, $p_2 \in nest(p_1)$ is mapped to a_1 of BusinessObjectA. Because $p_4 \in nest(p_1)$ is a complex parameter, it implies that its corresponding business object (i.e., BusinessObjectB) depends on p_1's corresponding business object (i.e., BusinessObjectA). This kind of relation is kept in ξ and it is represented using dashed arrow line in Fig. 2. Similarly, because $p_{10} \in nest(p_1)$ and it is an array of complex4, BusinessObjectA has a collection of BusinessObjectD and this indicates a one-to-many dependency relationship.

In this study, we derive the dependency relation according the hierarchical relation between complex parameters. In other words, some objects may depend on others because their corresponding parameters nest in other parameters. For example, ShipmentOrder has an attribute, which consists of a number of PackageLineItem objects. In this research, a dependent business object is called a weak object. Conversely, one that does not depend on any other business objects is called a strong object. A service s maintains M_s, which is a set of strong business objects implied by s. We can easily get strong objects from ξ. For example, $M = \{BusinessObjectA\}$ in the case of Fig. 2.

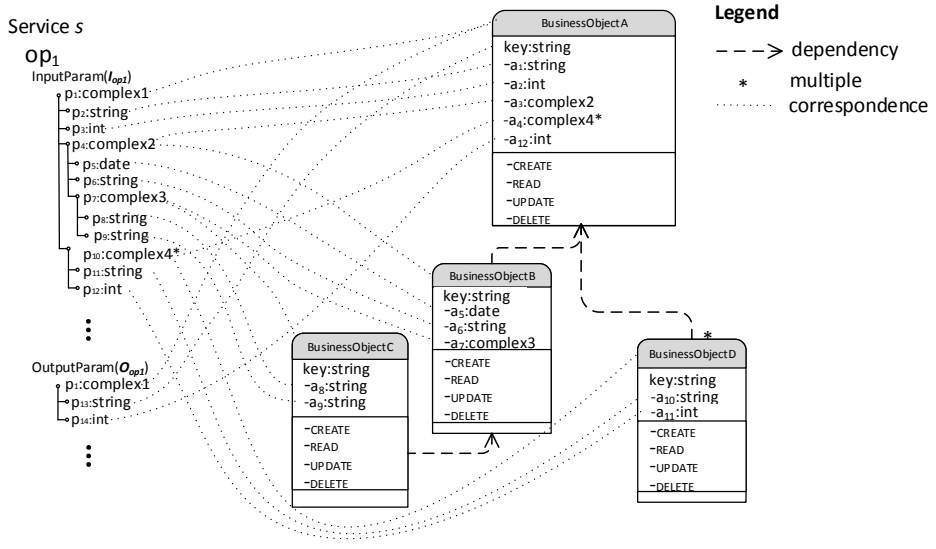

Fig. 2. A generic BO data model.

3.3 BO Data Model Derivation

Based on a given service specification such as a WSDL file, we firstly derive the BO data model. Algorithm 1 shows how a model is generated. Specifically, the algorithm takes a specification of service s as the input and produces the model (OP, BO, ξ) for s.

To semantically match a parameter with a business object, we assure the existence of an ontology to allow users to designate business objects for a particular context at design time. The business objects are stored in a business object repository $\mathcal{BO}$. At run-time, a parameter can be checked against the ontology to determine if there is a business object in $\mathcal{BO}$ semantically matches with the parameter. Specifically, the function OntologyCheck takes name ($\mathcal{N}(p)$) and type ($type(p)$) of a parameter, and the business object repository ($\mathcal{BO}$) as the inputs, and returns the matching business object in $\mathcal{BO}$. It will return nothing if there is no match found for the parameter p.

Algorithm 1 DeriveServiceDataModel

Require: $Spec_s$ (specification of service s), business objects repository $\mathcal{BO}$

```
/* Identify all the operations provided by s */
OP_s = IdentifyOP(Spec_s)
/* Identify business objects and their relations for the interface of s */
BO_s := ∅
ξ_s := ∅
for each op ∈ OP_s do
   for each p ∈ I(op) ∪ O(op) and γ(p) = complex do
      IdentifyBOandRelation(op, p, ⊥, BO_s, ξ_s, BO)
   end for
end for
return (OP_s, BO_s, ξ_s)
```

3.4 Service Operation Refactoring

To categorise operations provided by a service into four generic operations of business objects, we propose a mapping mechanism. As shown in Algorithm 3, the mechanism invokes each operation op that manipulates a business object bo (i.e., $op \in oprt(bo)$) and then analyses the input and output parameters to determine the category of op, i.e. whether op is to create, read, update or delete bo. Specifically, it compares these parameters with the key and attributes of current business object bo to check if there is any correspondence. For example, if the invocation requires some input parameters which are actually attributes of bo (i.e., $I_{min} \cap \mathcal{A}(bo) \neq \varnothing$) and it returns a value of $key(bo)$ (i.e., $key(bo) \in O_{rcv}$), the operation is for creating a bo instance. In this algorithm, the concept of business object is expanded by adding four sets: C, R, U, D and they are used to represent the set of operations for creating, reading, updating, and deleting a business object respectively. In essence, the algorithm analyses each operation $op \in oprt(bo)$ and groups it into one of the following sets: $C_{bo}, R_{bo}, U_{bo}, and D_{bo}$.

Algorithm 2 IdentifyBOandRelation

Require: operation op, (complex) parameter p, business object bo, set of objects BO, relations between objects ξ, business object repository $\mathcal{BO}$

```
/* Find a matching business object from the repository via ontology check */
bo' = OntologyCheck(N(p), type(p), BO)
/* Record the business object and derive the relation with its parent object */
if bo' ≠⊥ then
  AddToSet({bo'}, BO) /* i.e. BO = BO ∪ {bo'} */
  AddToSet({op}, oprt(bo'))
  AddToSet(nest(p), A(bo'))
  if bo ≠⊥ then
    AddToSet({(bo', bo)}, ξ)
  end if
  /* Recursively call this algorithm for each complex parameter nested in p */
  for each p' ∈ nest(p) and γ(p') = complex do
    IdentifyBOandRelation(op, p', bo', BO, ξ, BO)
  end for
end if
```

Algorithm 3 MapToCRUDoperations

Require: (Service) BO data model (OP, BO, ξ) of s

```
for each bo ∈ BO do
  C_bo := ∅
  R_bo := ∅
  U_bo := ∅
  D_bo := ∅
  for each op ∈ oprt(bo) do
    /* Select basic input parameters of operation op */
    I_min = GetMinInputParameters(op)
    /* Receive output parameters by invoking op using I_min */
    O_rcv = InvokeOperation(op, I_min)
    /* Map op to a CRUD operation based on I_min and O_rcv */
    if I_min ∩ A(bo) ≠ ∅ and key(bo) ∈ O_rcv then
      AddToSet({op}, C_bo)
    else if key(bo) ∈ I_min then
      if O_rcv ∩ A(bo) ≠ ∅ then
        AddToSet({op}, R_bo)
      else if I_min ∩ A(bo) ≠ ∅ then
        AddToSet({op}, U_bo)
      else if O_rcv = ∅ then
        AddToSet({op}, D_bo)
      end if
    end if
  end for
end for
```

The behavioural interfaces (i.e., protocols) of service describe a set of sequencing constraints. These constraints define legal order of messages by means of a finite-state grammar.

Definition 4 (Service protocol specification). A service protocol specification is a Petri net (Σ, T, F). T is a set of transitions that specify service operations, Σ a set of places that specify the pre- and post-conditions of service operations, and $F \subseteq (\Sigma \times T \cup T \times \Sigma)$ a set of flow relations that connect a (pre-)condition to an operation or an operation to a (post-)condition. □

This paper only deals with protocols of CRUD operations of each strong business object. As discussed in 3.2, a service s maintains a set M_s, which consists of strong business objects implied by s. So, for each $bo \in M_s$, we generate the protocol for its CRUD operations. For example, the protocol for CREAT operation defines the operations and their order to be called in order to create a business object. As an example, Algorithm 4 presents how a protocol for the generic CREAT operation of a strong business object is generated.

4 Validation and Implementation

4.1 Empirical Analysis

To demonstrate the necessity of this study, we empirically analyse the input interface of FedEx Shipping[1] and UPS Shipment[2] services to show the commonalities and differences. The findings show that there are around 93 pairs of common parameters across the FedEx and UPS shipping services. Each pair of parameters ontologically means the same thing. However, the correspondence is not obvious. Fig.3 presents a snapshot of the hierarchical correspondence. For example, seriveType at the first level in the FedEx shipping service matches with Shipment/Service/Code at the third level in UPS. DropoffType at the first level in the FedEx shipping service corresponds to a combination of HoldForPickUp and DropoffAtUPS at the third level in the UPS shipment service. At run-time, some of these common elements may not appear in one or another, so we can propose the missing parameters to the corresponding service. In addition to the similarity, there is also a significant amount (around 76 parameters) of differences between the two services. For instance, customsClearance is an input parameter in the FedEx service, but not in UPS. This being the case, a service consumer may reformulate requests with UPS shipment service to determine whether this newly understood feature (i.e., customs clearance) is supported in UPS.

4.2 Implementation

To validate the service interface synthesis framework, we have developed a prototype that introspects service interfaces and generates the business object data

[1] `http://www.fedex.com/templates/components/apps/wpor/secure/downloads/xml/Aug13/advanced/ShipService_v13.xml`

[2] `https://www.ups.com/upsdeveloperkit`

Algorithm 4 GeneratingProtocolForCreatingBO

Require: (Service) BO data model (OP, BO, ξ) of s, a strong $bo \in M_s$

```
/* Initialise the protocol specification (a Petri net) for creating bo */
Σ := {c_0, c_bo, c'_bo}
T := {τ_0}
F := {(c_0, τ_0)}
/* Find all the business objects that directly or indirectly depend on bo */
X_bo = {x ∈ BO\{bo} | (x, bo) ∈ ξ^+}
/* Map each business object and its read operation to a Petri net module */
for each x ∈ X_bo do
   rd_x = MapToTransition(op ∈ R_x) /* we assume R_x is a singleton */
   AddToSet({c_x, c'_x}, Σ)
   AddToSet({rd_x}, T)
   AddToSet({(c_x, rd_x), (rd_x, c'_x)}, F)
end for
/* Connect the above Petri net modules based on the logic flow of read operations */
ConnectReadOperationsForBOs(bo, X_bo, ξ, Σ, T, F)
/* Map each create operation of bo to a transition */
for each op ∈ C_bo do
   cr_op = MapToTransition(op)
   AddToSet({cr_op}, T)
end for
/* Identify the sequence of create operations for bo via introspection */
Y := C_bo
while Y ≠ ∅ do
   Z := Y
   repeat
      select op ∈ Z
      rsp = InvokeOperation(op, I(op))
      Z = Z \ {op}
   until rsp ≠ ⊥   /* i.e. until a positive response */
   /* Add this create operation and transition flow to the protocol specification */
   AddToSet({c_op, c'_op}, Σ)
   AddToSet({(c_op, cr_op), (cr_op, c'_op)}, F)
   Y = C_bo \ {op}
end while
return (Σ, T, F)
```

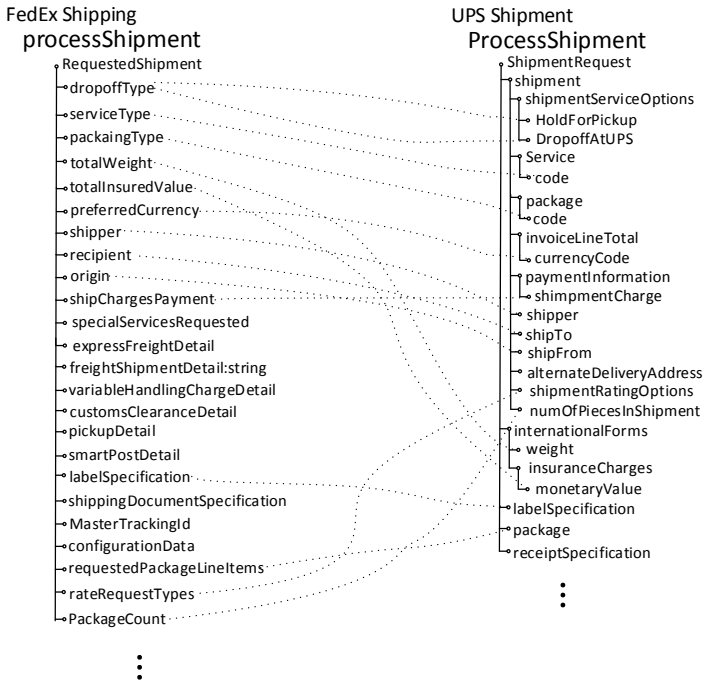

Fig. 3. The hierarchical alignment of input interfaces of shipment services between FedEx and UPS.

model and protocols for CRUD of each business object. This prototype is called service integration accelerator and it implements the algorithms presented in the previous sections.

In particular, there are four components in the *Service Integration Accelerator* as shown in Fig. 4. The *Business object editor* allows users to specify the business objects in a particular context. For example, in an interaction with the FedEx shipping service, a service consumer may specify OpenshipOrder as a business object. These business objects are stored in the *Business object ontology* and they are used to semantically match with parameters in service interfaces. The *Semantic matcher* component uses S-Match [9] to measure the semantic similarity between a parameter and business objects in the business object ontology. The *Service structural interface analyser* is the implementation of the Algorithm 1 in the section 3.3. This component generates business object data models for a service. These models are then used to synthesise service behavioural interfaces in the *Service behavioural interface introspector*. The introspector is an implementation of the Algorithm 3 and Algorithm 4 in the section 3.4. The structural and behavioural interfaces derived are used by the *Service interface backtracker* to evolve service requests. Based on the backtracking result, service interfaces can be normalised by the *Service interface normaliser*. The normalised interfaces are then stored as references in the *Service interface repository* for adaptation and future reuse.

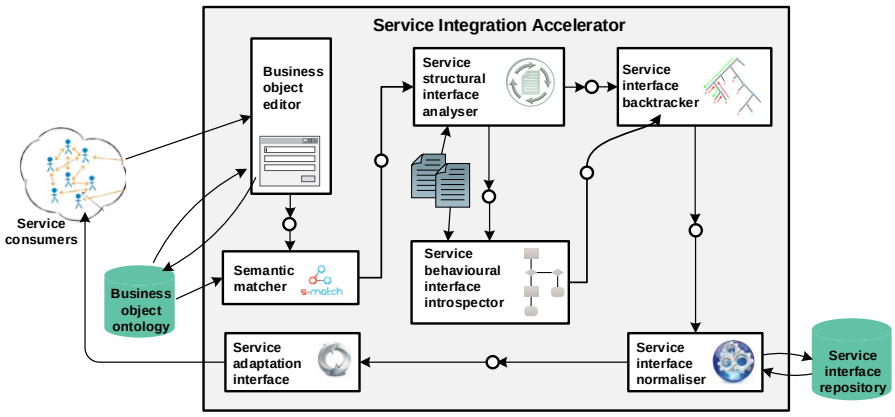

Fig. 4. The overview of the service integration accelerator.

We applied the prototype to the FedEx open shipping service. Its WSDL specification is available online[3]. Fig.5 shows a screenshot of the generated data model and this is the result of Algorithm 1 in Sect. 3.3. The business objects derived are OpenshipOrder, PackageLineItem, Payment, Shipper, Recipient, CustomsClearance, and Label. The protocols of creating an OpenshipOrder is presented as a Petri nets state transition model as shown in Fig.6 and this is the result of Algorithm 4 in Sect. 3.4. This model shows the operations and the order to be executed in order to create an OpenshipOrder. For example, following the initial state, all business objects that depend on the OpenshipOrder are prepared. This is achieved by calling READ (R) operation of business objects involved such as Customs and Shipper. Once these dependent business objects are in place, the operation "createOpenShipment" can be called to start to create an OpenshipOrder, this is followed by the operation "addPackagesToOpenShipment" to add PackageLineItems. Once all items are added, the operation "confirmOpenShipment" is called to confirm the open ship order creation. In this process, the state (i.e., the pre- and post-condition) of OpenshipOrder is transferred from "Initial" to "Confirmed" and the intermediate states are "OpenshipOrderCreated" and "packageAdded".

5 Related Work

Various approaches for service adaptation have been proposed in the existing studies over recent years. Pattern-based approaches categorise mismatches into a number of patterns and address them with the corresponding resolution patterns. Service mismatch patterns [10] are defined as a way of capturing and resolving differences. Rule-based approaches [3] address mismatches using logic by applying rules and conditions, and use algorithms to produce adapters. Planner-based approaches [11] use visualization tools to analyze and resolve service mismatches at design time. However, most of these existing adaptation techniques

[3] `http://www.fedex.com/templates/components/apps/wpor/secure/downloads/xml/Aug13/advanced/OpenShipService_v5.xml`

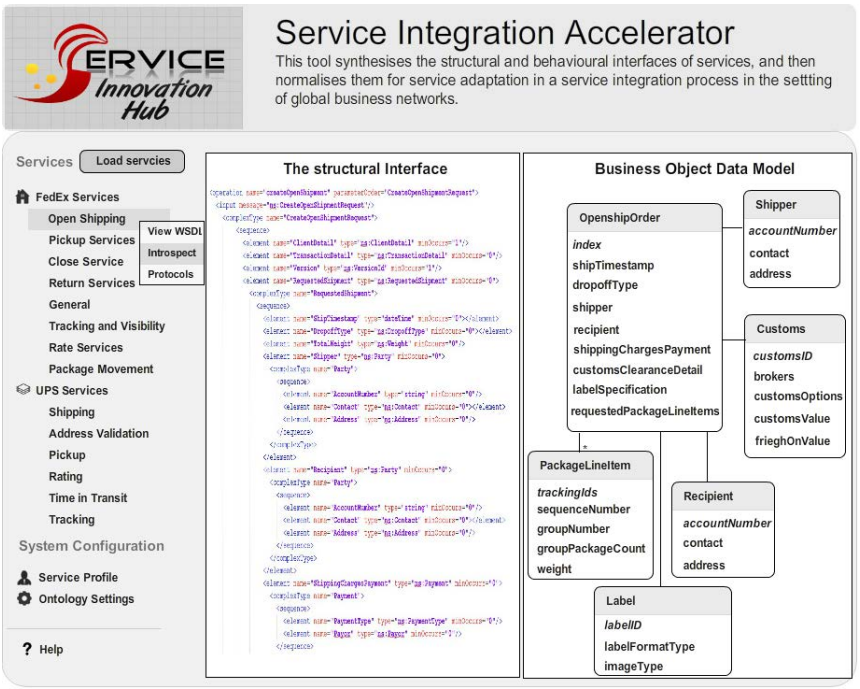

Fig. 5. Interface analysis on the FedEx Open shipping.

assume that service behavioural interfaces are available all the time. This study argues that, in the setting of business networks, many service providers do not provide behavioural interface specifications. Therefore, it is important to synthesis behavioural interfaces before adapting them. In addition, the up-to-date research in service adaptation has not addressed the complex and overloaded service interfaces. For example, they do not directly address self-learning and service interface synthesis.

Service protocol synthesis addresses generation of service behavioural interfaces based on various techniques including static analysis, interaction log mining, and service composition. Static analysis techniques involve analysing structural and behavioural specification of services. For example, Cavallaro et al. [12] proposed an approach to synthesise service protocols based on WSDL interface. However, this approach only can generate protocols of one service. That is to say, only invocation sequences within one service WSDL specification can be identified and it has not addressed protocol synthesis in a conversational context where more than one services (each service has a WSDL specification) are involved. The mining approach [8] heavily replies on service interaction logs, and it is not applicable if service interaction logs are not provided. In a service composition, the common problem being addressed is "how to automatically generate a new target service protocol by reusing some existing ones" [13]. However, this technique assumes the interfaces of individual services involved in a composition are available.

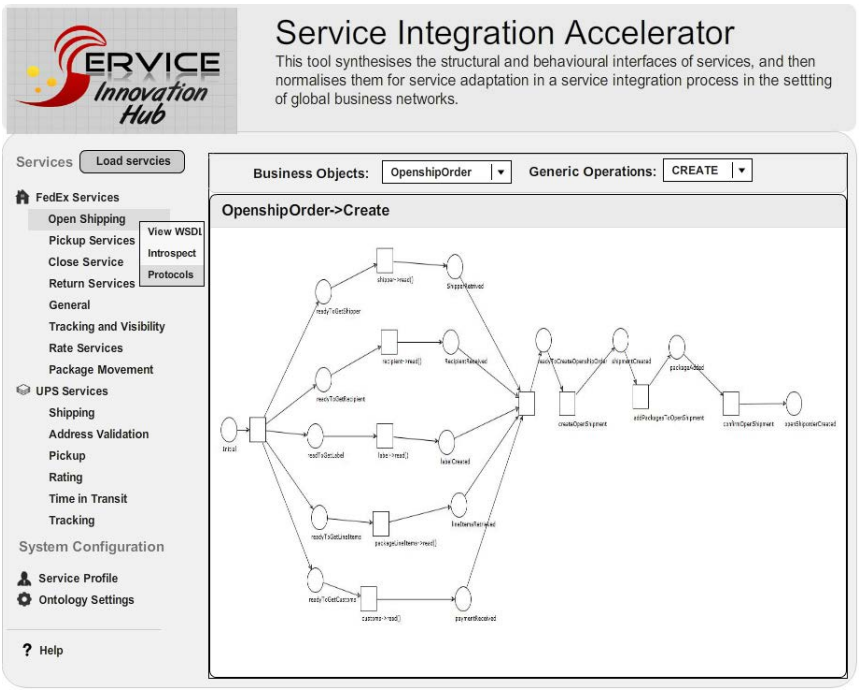

Fig. 6. Generate Fedex OpenshipOrder CREATE protocols.

6 Conclusion

This paper presents a service interface synthesis framework for addressing the service adaptation challenges in the context of open and diffuse setting of global business networks. We described patterns motivating novel requirements for service interface synthesis and the key components of the framework, detailing service interface synthesis. We also validated the framework using complex services drawn from the logistic domain. The study demonstrated that the business object based synthesis technique is an effective solution to determining valid invocations made against large, overloaded operations in interfaces inherent with multiple service variants. Future work will focus on improving the synthesis technique with the two open issues. The first is that we will use the structured businss object data model to compose different invocations and then validate them using a trial/error introspection mechanism. The second is that we have not examined the case when multiple keys of business objects returned in a response of an invocation. The Algorithm 3 in Sect. 3.4 only considers one key of one business object. Specialisation of business objects is another issue that needs to be considered when we derive business object data model. That is to say, a sub business object may be revealed from interfaces of services. Local introspection for each CRUD operation can be a solution to deriving sub business objects. We will examine these issues in the next step of this study. We will also develop a backtracking mechanism and focus on yielding generalised interfaces

for similar services, by abstracting their structural and behavioural interfaces, in order to allow for these to be normalised (i.e. merged) into general definitions which promote adaptation reuse for similar services. Finally, we will extend the prototype to support local introspection, backtracking and normalisation and this tool will be openly used and validated on the Internet.

References

1. Goethals, F., Vandenbulcke, J., Lemahieu, W.: Using web services in business-to-business integration. In: Electronic Commerce: Concepts, Methodologies, Tools, and Applications. IGI Global, Hershey, PA, USA (2008)
2. Mateescu, R., Poizat, P., Salaun, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Transactions on Software Engineering **38**(4) (2012) 755–777
3. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of the 16th international conference on World Wide Web (WWW'07), New York, NY, USA, ACM (2007) 993–1002
4. Wei, F., Barros, A.P., Ouyang, C.: Analysis of multilateral service conversations in global business networks. (2013)
5. Barros, A., Dumas, M., Hofstede, A.: Service interaction patterns. In Aalst, W., Benatallah, B., Casati, F., Curbera, F., eds.: Business Process Management. Volume 3649 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 302–318
6. Oberle, D., Barros, A., Kylau, U., Heinzl, S.: A unified description language for human to automated services. Information Systems **38**(1) (2013) 155 – 181
7. Oaks, P., Hofstede, t.A.: Guided interaction: A mechanism to enable ad hoc service interaction. Information Systems Frontiers **9**(1) (March 2007) 29–51
8. Motahari-Nezhad, H., Saint-Paul, R., Benatallah, B., Casati, F.: Protocol discovery from imperfect service interaction logs. In: IEEE 23rd International Conference on Data Engineering, 2007. (2007) 1405–1409
9. Giunchiglia, F., Shvaiko, P., Yatskevich, M.: S-match: an algorithm and an implementation of semantic matching. In Bussler, C., Davies, J., Fensel, D., Studer, R., eds.: The Semantic Web: Research and Applications. Volume 3053 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 61–75
10. Li, X., Fan, Y., Madnick, S., Sheng, Q.Z.: A pattern-based approach to protocol mediation for web services composition. Information and Software Technology **52**(3) (March 2010) 304–323
11. Dumas, M., Spork, M., Wang, K.: Adapt or perish: Algebra and visual notation for service interface adaptation. In Dustdar, S., Fiadeiro, J., Sheth, A., eds.: Business Process Management. Volume 4102 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2006) 65–80
12. Cavallaro, L., Di Nitto, E., Pelliccione, P., Pradella, M., Tivoli, M.: Synthesizing adapters for conversational web-services from their wsdl interface. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '10, New York, NY, USA, ACM (2010) 104–113
13. Ragab Hassen, R., Nourine, L., Toumani, F.: Protocol-based web service composition. In: Proceedings of the 6th International Conference on Service-Oriented Computing. ICSOC '08, Berlin, Heidelberg, Springer-Verlag (2008) 38–53